Events

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`22/27`
Semester	`Sommersemester 2025`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	Publish-Subscribe Prinzip, Events in C#, generische Events
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/22_Events.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich

Allgemeine Hinweise

0. Kennen Sie Ihren Editor und dessen Shortcuts!

https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf

1. Verwenden Sie einen *Code Formater*, der Ihnen bei der Restrukturierung Ihres Codes hilft!

2. Evaluieren Sie die Hinweise der Code Analyse sorgfältig, entwerfen Sie ggf. eigene Regeln.

https://learn.microsoft.com/de-de/dotnet/fundamentals/code-analysis/overview?tabs=net-8

Nachgefragt

In der letzten Veranstaltung fragte einer von Ihnen wie der selektive Zugriff auf die MultiCastDelegaten realisieren kann. Zur Erinnerung MulticastDelegate verfügt über eine verknüpfte Liste von Delegaten, die als Aufruf Liste bezeichnet wird und aus einem oder mehreren-Elementen besteht. Wenn ein Multicast Delegat aufgerufen wird, werden die Delegaten in der Aufruf Liste synchron in der Reihenfolge aufgerufen, in der Sie angezeigt werden.

```
Mulitcast Invocation List
```

`Add`	`Multiply`	`Multiply`	`Divide`	`Add`	`Divide`

.

Frage: Wie können wir unsere MultiCastDelegaten verwalten und selektiv auf einzelne Elemente zugreifen?

vgl. https://learn.microsoft.com/en-us/dotnet/api/system.delegate.getinvocationlist?view=net-8.0

MultiCast

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class Program
{
  public delegate int Calc(int x, int y);

  static int Add(int x, int y){
      Console.WriteLine("x + y");
      return x + y;
  }

  static int Multiply(int x, int y){
      Console.WriteLine("x * y");
      return x * y;
  }

  static int Divide(int x, int y){
      Console.WriteLine("x / y");
      return x / y;
  }

  public static void Main(string[] args){
    Calc computer = Add;
    computer += Multiply;
    computer += Multiply;
    computer += Divide;
    computer += Add;
    computer += Divide;
    computer -= Add;
    Console.WriteLine("Zahl von eingebundenen Delegates {0}",
                      computer.GetInvocationList().GetLength(0));

    // Individueller Aufruf der einzelnen Einträge
    var x = computer.GetInvocationList();
    Console.WriteLine("Typ der Invocation List {0}", x.GetType());
    Console.WriteLine(x[0].DynamicInvoke(1,2));
    Console.WriteLine(x[1].DynamicInvoke(3,5));

    // Übergreifender Aufruf aller Einträge
    Console.WriteLine(computer(40,8));
  }
}
```

```
Zahl von eingebundenen Delegates 5
Typ der Invocation List System.Delegate[]
x + y
3
x * y
15
x + y
x * y
x * y
x / y
x / y
5
Zahl von eingebundenen Delegates 5
Typ der Invocation List System.Delegate[]
x + y
3
x * y
15
x + y
x * y
x * y
x / y
x / y
5
```

Frage: Wie ließe sich das Codebeispiel verbessern? Wie war das gleich noch `Func` oder `Action`?

Wiederholung

Wie war das noch mal, welche Elemente (Member) zeichnen einen Klasse unter C# aus?

Bezeichnung
Konstanten
Felder
Methoden
Eigenschaften
Indexer
Ereignisse
Operatoren
Konstruktoren
Finalizer
Typen

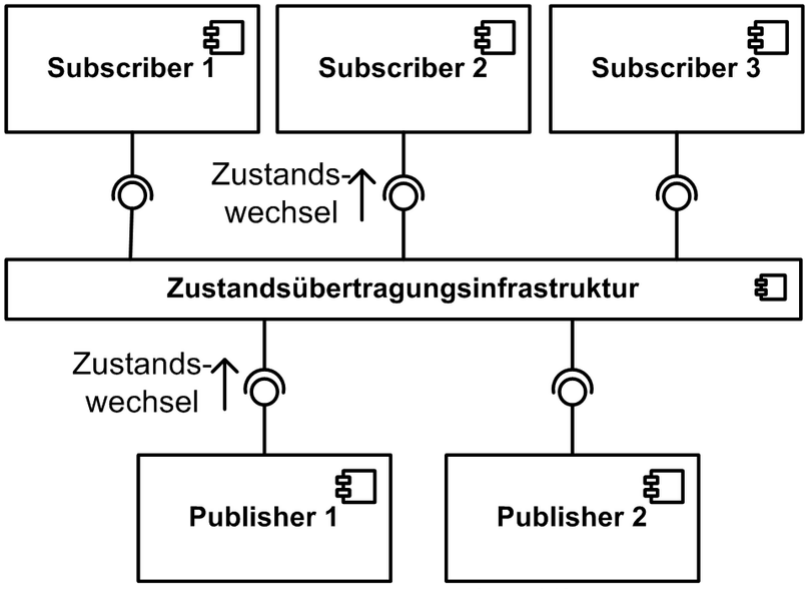
Bezeichnung	Bedeutung
Konstanten	Konstante Werte innerhalb einer Klasse
Felder	Variablen der Klasse
Methoden	Funktionen, die der Klasse zugeordnet sind
Eigenschaften	Mechanismen zum Lesen und Schreiben auf geschützter Variablen
Indexer	Spezifikation eines Indexoperators für die Klasse
Ereignisse	?
Operatoren	Definition von eigenen Symbolen für die Arbeit mit Instanzen der Klasse
Konstruktoren	Bündelung von Aktionen zur Initialisierung der Klasseninstanzen
Finalizer	Aktionen, die vor dem "Löschen" der Instanz ausgeführt werden
Typen	Geschachtelte Typen, die innerhalb der Klasse definiert werden

Motivation und Idee der Events

Was haben wir mit den Delegaten erreicht? Wir sind in der Lage aus einer Klasse, auf Methoden (einer anderer Klassen) zurückzugreifen, über die wir per Referenz informiert wurden. Die aufgerufene Methode wird der aufrufenden Klasse über einen Delegaten bekannt gegeben. Es erfolgt eine Typprüfung der Parameter.

Damit sind die beiden Klassen nur über eine Funktionssignatur miteinander "lose" gekoppelt. Welche Konzepte lassen sich damit umsetzen?

Publish-Subscribe Prinzip



Publish Subscribe Kommunikationsmodell [HKoziolek]

Publish-Subscribe (Pub / Sub) ist ein Nachrichtenmuster, bei dem Publisher Nachrichten an Abonnenten senden. In der Softwarearchitektur bietet Pub / Sub-Messaging Ereignisbenachrichtigungen für verteilte Anwendungen, insbesondere für Anwendungen, die in kleinere, unabhängige Bausteine entkoppelt sind.

Das Publish/Subscribe-Nachrichtenmodell hat folgende Vorteile:

- Der Publisher braucht gar nicht zu wissen, wer Subscriber ist. Es erfolgt keine explizite Adressierung
- Der Subscriber ist vom Publisher entkoppelt, er empfängt nur die Nachrichten, die für ihn auch relevant sind.
- Der Subscriber kann sich jederzeit aus der Kommunikation zurückziehen oder ein anderes Topic subskribieren. Auf den Publisher hat das keine Auswirkung.
- Damit ist die Messaging-Topologie dynamisch und flexibel. Zur Laufzeit können neue Subscriber hinzukommen.

C# etabliert für die Nutzung der Pub-Sub Kommunikation *Events*. Dies sind spezielle Delegate-Variablen, die mit den Schlüsselwort `event` als Felder von Klassen deklariert werden.

[HKoziolek] Wikipedia Publish/Subscribe Architekturstil für Software, Autor HKoziolek, https://de.wikipedia.org/wiki/Datei:Publish-Subscribe-Architekturstil.png

Events in C#

Der Publisher ist eine Klasse, die eine Delegate-Variable enthält. Der Publisher entscheidet damit darüber, wann Nachrichten versandt werden. Auf der anderen Seite finden sich die Subscriber-Methoden, die ausgehend vom aktivierten Delegaten im Publisher zur Ausführung kommen. Ein Subscriber hat keine Kenntnis von anderen Subscribern. Events sind ein Feature aus C# dass dieses Pattern formalisiert.

Merke: Ein Event ist ein Klassenmember, dass die Features des Delegatenkonzepts nutzt, um eine Publisher-Subscribe Interaktion zu realisieren.

Im einfachsten Fall lässt sich das Event-Konzept folgendermaßen anwenden:

Event.cs

```
// Schritt 1
// Wir definieren einen Delegat, der das Format der Subscriber Methoden
// (callbacks) spezifiziert - in diesem Beispiel ohne Parameter
// Hier wird ein nicht-generischer Handler genutzt.
public delegate void varAChangedHandler();

// Schritt 2
// Wir integrieren ein event in unser Publisher Klasse, dass den Delegaten
// abbildet
public class Publisher{
  public event varAChangedHandler OnAChangedHandler;

  // Schritt 3
  // Wir implementieren das "Feuern" des Events
  public void magicMethod(){
    if (oldA != newA) OnAChangedHandler();
  }
}

// Schritt 4
// Implementieren des Subscribers - in diesem Fall wurde eine separate Klas
// gewählt.
public class Subscriber{
  public void m_OnPropertyChanged(){
     Console.WriteLine("A was changed!");
  }
}

// Schritt 5
// "Einhängen" der Subscriber Methode in den Publisher-Delegate
public static void Main(string[] args){
  Publisher myPub = new Publisher();
  Subscriber mySub = new Subscriber();
  myPub.OnAChangedHandler += new varAChangedHandler(mySub.m_OnPropertyChang
  myPub.magicMethod();
}
```

Welche Konsequenzen ergeben sich daraus?

- Der Publisher bestimmt, wann ein Ereignis ausgelöst wird. Die Subscriber bestimmen, welche Aktion als Reaktion auf das Ereignis ausgeführt wird.
- Es ist eine 1:n Relation. Ein Ereignis entstammt einem Publisher, kann aber mehrere Subscriber adressieren.
- Achtung: Ereignisse, die keine Subscriber haben werfen NullReferenzException aus, wenn sie ausgelöst werden (OnAChangedHandler?.Invoke();).
- Ereignisse werden in der Regel verwendet, um Benutzeraktionen wie Mausklicks oder Menüauswahlen in GUI-Schnittstellen zu signalisieren.
- In der .NET Framework-Klassenbibliothek basieren Ereignisse auf dem EventHandler-Delegaten und der EventArgs-Basisklasse.

Was passiert hinter den Kulissen?

Wenn wir ein `event` deklarieren

```
public class Publisher{
  public event VarAChangedHandler AChanged;
}
```

passieren folgende 3 Dinge:

1. Der Complier erzeugt einen privaten Delegaten mit sogenannten Event Accessoren (add und remove).

```
VarAChangedHandler aChanged; // private Delegate
public event VarAChangedHandler AChanged
{
  add {aChanged += value;}
  remove {aChanged -= value;}
}
```

2. Der Compiler sucht innerhalb der Publisher Klasse nach Referenzen auf AChanged und lenkt diese auf das private Delegate um.
3. Der Compiler mappt alle += bzw. -= Operationen außerhalb auf die Zugriffsmethoden add bzw. remove.

Ok, nett, aber das würde ich gern an einem Beispiel sehen!

Beispiel 1

Das Beispiel vereinfacht das Vorgehen, in dem es Publisher und Subscriber in einer Funktion zusammenfasst. Damit kann auf das Event uneingeschränkt zurückgegriffen werden. Dazu gehört auch, dass das Event mit `Invoke` ausgelöst wird.

MinimalEvent.cs

```
1  using System;
2  using System.Reflection;
3  using System.Collections.Generic;
4
5  public delegate void DelEventHandler();
6
7  class Program
8  {
9      public static event DelEventHandler myEvent;
10     public static void Main(string[] args){
11         myEvent += new DelEventHandler(Fak1);
12         myEvent += Fak2;
13         myEvent += () => {Console.WriteLine("Fakultät 3");};
14         myEvent.Invoke();
15     }
16
17     static void Fak1()
18     {
19         Console.WriteLine("Fakultät 1");
20     }
21
22     static void Fak2()
23     {
24         Console.WriteLine("Fakultät 2");
25     }
26 }
```

```
Fakultät 1
Fakultät 2
Fakultät 3
Fakultät 1
Fakultät 2
Fakultät 3
```

Beispiel 2

StockExchange

```
using System;
using System.Reflection;
using System.Collections.Generic;

public delegate void DelPriceChangedHandler();

public class Stock{
    decimal price = 5;
    public event DelPriceChangedHandler OnPropertyPriceChanged;
    public decimal Price{
      get { return price; }
      set { if (price != value){
                price = value;
                if (OnPropertyPriceChanged != null){
                    OnPropertyPriceChanged();
                }
            }
          }
    }
}

public class MailService{
  public static void stock_OnPropertyChanged(){
      Console.WriteLine("MAIL - Price of stock was changed!");
  }
}

public class Logging{
  public static void stock_OnPropertyChanged(){
      Console.WriteLine("LOG  -  Price of stock was changed!");
  }
}

class Program {
    public static void Main(string[] args){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyPriceChanged += MailService
          .stock_OnPropertyChanged;
        GoogleStock.OnPropertyPriceChanged += Logging
          .stock_OnPropertyChanged;
        Console.WriteLine("We change the stock price now!");
        GoogleStock.Price = 10;
    }
}
```

```
We change the stock price now!
MAIL - Price of stock was changed!
LOG  -  Price of stock was changed!
We change the stock price now!
MAIL - Price of stock was changed!
LOG  -  Price of stock was changed!
```

Ja, aber ... Was unterscheidet eine Delegate von einem Event? Was würde passieren, wenn wir das Key-Wort aus dem Code entfernen?

Die Implementierung wäre nicht so robust, da folgende Möglichkeiten offen ständen und damit :

Eingriff	Bedeutung	Verhindert
`GoogleStock.OnPropertyPriceChanged = null;`	Löscht alle Callback-Handler	ja
`GoogleStock.OnPropertyPriceChanged = DelPriceChangedHandler(MailService.stock_OnPropertyChanged);`	Setzt einen einzigen Handler (und löscht alle anderen)	ja
`GoogleStock.OnPropertyPriceChanged.Invoke();`	Auslösen des Events innerhalb eines Subscribers	ja

Das Weglassen des Schlüsselwortes event untergräbt das Prinzip der Kapselung und führt leicht zu unbeabsichtigtem Verhalten wie z. B. doppelte Aufrufe.

Das event-Keyword dient als Zugriffsmodifizierer für Delegaten und stellt sicher, dass:

- Nur der Publisher das Event auslösen kann,
- Externe Objekte keinen direkten Zugriff auf den Delegaten selbst bekommen.

Was fehlt Ihnen an der Implementierung?

Richtig, die Möglichkeit auf die Daten zurückzugreifen.

Events - Praktische Implementierung

Die Möglichkeit Parameter strukturiert und wiederverwendbar zu übergeben und entsprechend generische EventHandler zu integrieren.

Parameter

Welche Informationen sollten an die Subscriber weitergereicht werden? Zum einen der auslösende Publisher (Wer ist verantwortlich?) und ggf. weitere Daten zum Event (Warum ist die Information eingetroffen?).

Im Beispiel konzentrieren wir uns auf die Default-Delegates, die Bestandteil der .NET Umgebung ist

Delegate	Aufruf	Link
`EventHandler Delegate`	`void EventHandler(object sender, EventArgs e)`	Link
`EventHandler<TEventArgs> Delegat`	`EventHandler<TEventArgs>(object? sender, TEventArgs e);`	Link

StockExchangeII

```
using System;
using System.Reflection;
using System.Collections.Generic;

//Brauchen wir nicht mehr, vielmehr wird auf einen vordefinierten
  EventHandler
//zurückgegriffen.
//public delegate void DelPriceChangedHandler();
public class Stock{
    decimal price = 5;
    // Hier ersetzen wir unser Delegate "DelPriceChangedHandler" durc
    // Standard-Typ EventHandler
    public event EventHandler OnPropertyPriceChanged;
    public decimal Price{
      get { return price; }
      set { if (price != value){
                price = value;
                if (OnPropertyPriceChanged != null){
                    OnPropertyPriceChanged(this, EventArgs.Empty);
                }
            }
          }
    }
}

public class MailService{
  public static void stock_OnPropertyChanged(object sender, EventArgs
      Console.WriteLine("MAIL - Price of stock was changed!");
      Console.WriteLine("Wer ruft? - {0}", sender);
      Console.WriteLine("Werte? - {0}", e);
  }
}

class Program {
    public static void Main(){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyPriceChanged += new
                EventHandler(MailService.stock_OnPropertyChanged);
        Console.WriteLine("We changed the stock price now!");
        GoogleStock.Price = 10;
    }
}
```

```
We changed the stock price now!
MAIL - Price of stock was changed!
Wer ruft? - Stock
Werte? - System.EventArgs
We changed the stock price now!
MAIL - Price of stock was changed!
Wer ruft? - Stock
Werte? - System.EventArgs
```

Generic Events

Nun wollen wir einen Schritt weitergehen und spezifischere Informationen mit dem Event an die Subscriber weiterreichen. Dafür implementieren wir eine von `EventArgs` abgeleitete Klasse.

```
public class PriceChangedEventArgs : EventArgs
{
    public decimal Difference { get; set; }
}
```

Diese soll den Unterschied zwischen altem und neuen Preis umfassen, damit der Subscriber die Relevanz der Information beurteilen kann.

Damit brauchen wir aber auch ein neues Delegate für unser nun nicht mehr Standard Event

```
void EventHandler(object sender, PriceChangedEventArgs e)
```

Um diese Anpassungen beim Datentyp zu realisieren existiert bereits eine generischen Form von EventHandler mit der Signatur

```
public delegate void EventHandler<TEventArgs>(object source,
                                               TEventArgs e)
                                               where TEventArgs: EventArgs;
```

StockExchangeIII

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class Stock{
    decimal price = 5;
    public event EventHandler<PriceChangedEventArgs> OnPropertyPriceC
      ;
    public decimal Price{
      get { return price; }
      set { if (price != value){
                if (OnPropertyPriceChanged != null){
                    PriceChangedEventArgs myEventArgs = new
                      PriceChangedEventArgs();
                    myEventArgs.Difference = price - value;
                    OnPropertyPriceChanged(this, myEventArgs);
                }
                price = value;
            }
          }
    }
}

public class PriceChangedEventArgs : EventArgs
{
    public decimal Difference { get; set; }
}

public class MailService{
  public static void stock_OnPropertyChanged(object sender,
    PriceChangedEventArgs e){
      Console.WriteLine("MAIL - Price of stock was changed!");
      Console.WriteLine("Wer ruft? - {0}", sender);
      Console.WriteLine("Preisänderung? - {0}", e.Difference);
  }
}

class Program {
    public static void Main(string[] args){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyPriceChanged += new
                EventHandler<PriceChangedEventArgs>(MailService
                  .stock_OnPropertyChanged);
        Console.WriteLine("We manipulate the stock price now!");
        GoogleStock.Price = 10;
    }
}
```

```
We manipulate the stock price now!
MAIL - Price of stock was changed!
Wer ruft? - Stock
Preisänderung? - -5
We manipulate the stock price now!
MAIL - Price of stock was changed!
Wer ruft? - Stock
Preisänderung? - -5
```

Events und Ausnahmebehandlung

> Welche Rückmeldung hätten Sie mit Blick auf die Flexibilität des Codes an einen Mitstreiter?

MinimalEvent.cs

```
using System;
using System.Reflection;
using System.Collections.Generic;

public delegate void MyEventHandler(string message);

public class Publisher
{
    public event MyEventHandler MyEvent;
    public void RaiseEvent(string message)
    {
        MyEvent?.Invoke(message);
    }
}

public class Subscriber1
{
    //int x = 100;
    //int divider = 0;
    public void OnEventRaised(string message)
    {
        Console.WriteLine($"Sub 1 - Event received: {message}");
        //Console.WriteLine($"Sub 1 - Event received: {x / divider}")
    }
}

public class Subscriber2
{
    public void OnEventRaised(string message)
    {
        Console.WriteLine($"Sub 2 - Event received: {message}");
    }
}

class Program {
    public static void Main(string[] args){
        var publisher = new Publisher();
        var subscriber1 = new Subscriber1();
        publisher.MyEvent += subscriber1.OnEventRaised;
        var subscriber2 = new Subscriber2();
        publisher.MyEvent += subscriber2.OnEventRaised;
        publisher.RaiseEvent("Hallo Welt");
    }
}
```

```
Sub 1 - Event received: Hallo Welt
Sub 2 - Event received: Hallo Welt
Sub 1 - Event received: Hallo Welt
Sub 2 - Event received: Hallo Welt
```

Solution.cs

```
using System;
using System.Reflection;
using System.Collections.Generic;

// nicht mehr benötigt !
// public delegate void MyEventHandler(string message);

public class CustomEventArgs : EventArgs
{
    public string Message { get; }
    public CustomEventArgs(string msg)
    {
        Message = msg;
    }
}

public class Publisher
{
    public event EventHandler<CustomEventArgs> MyEvent;
    public void RaiseEvent(string message)
    {
      foreach (var handler in MyEvent.GetInvocationList())
      {
          try
          {
              handler.DynamicInvoke(this, new CustomEventArgs(message
          }
          catch (Exception ex)
          {
              Console.WriteLine($"Exception caught: {ex.Message}");
          }
      }
    }
}

public class Subscriber1
{
    int x = 100;
    int divider = 0;
    public void OnEventRaised(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"Sub 1 - Event received: {e.Message}");
        Console.WriteLine($"Sub 1 - Event received: {x/divider}");
    }
}

public class Subscriber2
{
```

```
    public void OnEventRaised(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"Sub 1 - Event received: {e.Message}");
    }
}

class Program {
    public static void Main(string[] args){
        var publisher = new Publisher();
        var subscriber1 = new Subscriber1();
        publisher.MyEvent += subscriber1.OnEventRaised;
        var subscriber2 = new Subscriber2();
        publisher.MyEvent += subscriber2.OnEventRaised;
        publisher.RaiseEvent("Hallo Welt");
    }
}
```

```
Sub 1 - Event received: Hallo Welt
Exception caught: Exception has been thrown by the target of an
invocation.
Sub 1 - Event received: Hallo Welt
Sub 1 - Event received: Hallo Welt
Exception caught: Exception has been thrown by the target of an
invocation.
Sub 1 - Event received: Hallo Welt
```

Anpassung der Subscribe/Unsubscribe Methoden

In spezifischen Fällen kann es notwendig sein, die Subscriber-Methoden zu adaptieren.

```
private MyEventHandler _myEvent;

public event MyEventHandler MyEvent
{
    add
    {
        Console.WriteLine("Subscriber added");
        _myEvent += value;
    }
    remove
    {
        Console.WriteLine("Subscriber removed");
        _myEvent -= value;
    }
}
```

Anonyme / Lambda Funktionen als Subscriber

MinimalEvent.cs

```cs
using System;
using System.Reflection;
using System.Collections.Generic;

public class CustomEventArgs : EventArgs
{
    public string Message { get; }
    public CustomEventArgs(string msg)
    {
        Message = msg;
    }
}

public class Publisher
{
    public event EventHandler<CustomEventArgs> MyEvent;
    public void RaiseEvent(string message)
    {
        MyEvent?.Invoke(this, new CustomEventArgs(message));
    }
}

class Program {
    public static void Main(){
        var publisher = new Publisher();

        // Using an anonymous method
        publisher.MyEvent += delegate(object sender, CustomEventArgs
        {
            Console.WriteLine($"Anonyme Method received: {args.Messag
        };

        // Or using a lambda expression
        publisher.MyEvent += (sender, args) => Console.WriteLine($"La
          expression received: {args.Message}");
        publisher.RaiseEvent("Hallo Welt");
    }
}
```

```
Anonyme Method received: Hallo Welt
Lambda expression received: Hallo Welt
Anonyme Method received: Hallo Welt
Lambda expression received: Hallo Welt
```

Eine sehr anschauliche Darstellung dazu findet sich unter https://gehirnwindung.de/categories/csharp/tanz-den-lambda-mit-mir

Grafische Nutzer Interfaces

siehe Codebeispiel `wpf_sharp` im Projektordner